“Bottom up” hybrid reimplementation of the *Jayenne* Project’s Implicit Monte Carlo* for Roadrunner

*a.k.a. Milagro

Tim Kelley, CCS-2

Roadrunner Technical Seminar Series

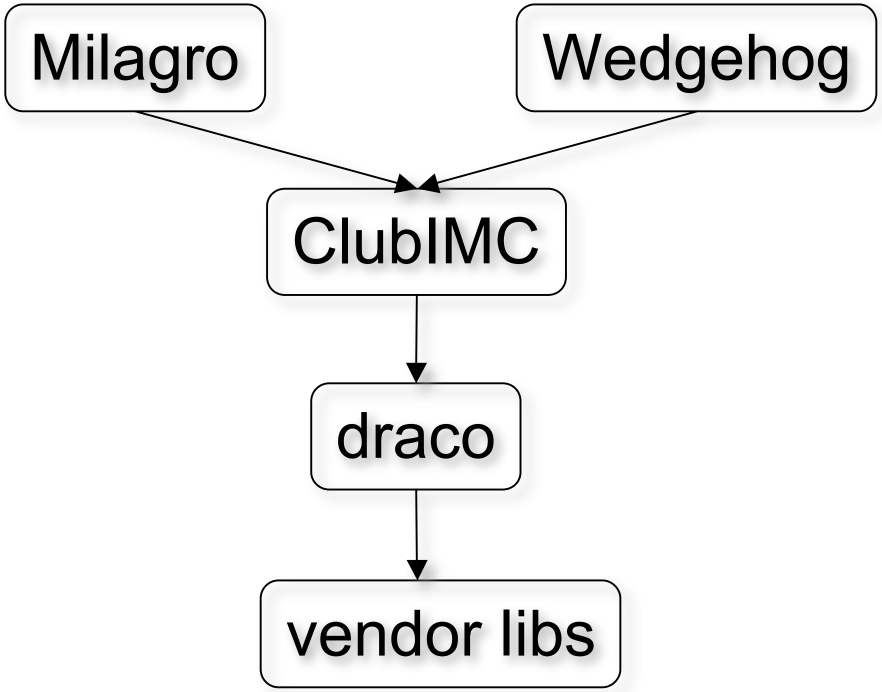
April 24, 2008

Acknowledgements

- CCS-2 IMC team: Buksas, Densmore, Fryer, Hungerford, Mosher, Urbatsch (, & Evans)
- RR-Algs team: Bergen, Desai, Inman, Henning, Koch, McCormick, Mohd-Yusof, Swaminarayan, Turner, ...
- CCS-1 PAL team: Kerbyson and Lang
- IBM: Wright and Brokenshire

The *Jayenne* Project provides Implicit Monte Carlo simulation capability.

- ClubIMC: “Component library used by IMC”
  - the core product
- Milagro: rad-only driver applications
  - for testing, method dev, etc.
- Wedgehog: Fortran callable interface library



Milagro

Wedgehog

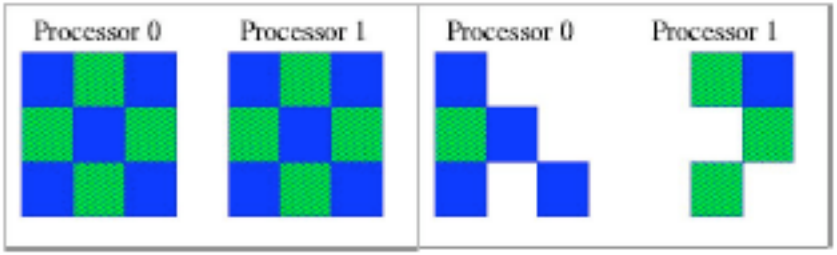
ClubIMC

draco

vendor libs

Implicit Monte Carlo simulates thermal X-ray transport for time dependent, nonlinear problems

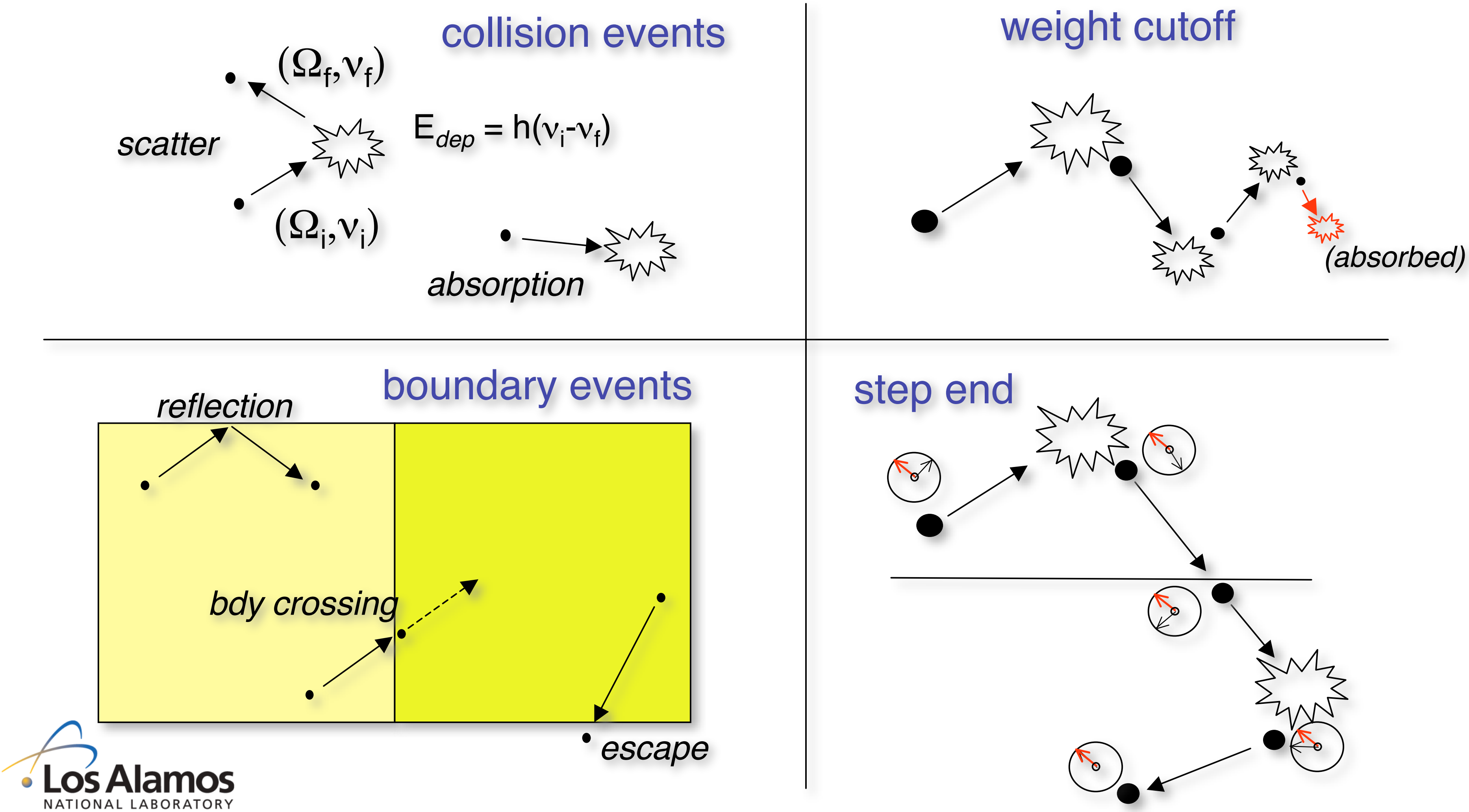
- Fleck & Cummings time discretization
  - effective scattering even in purely absorbing media
  - “Implicit” refers to the time discretization, not the Monte Carlo
- object-oriented, generic C++
  - templated on mesh type, freq. type, particle type.
- particles transported in 3D, meshes articulated in 1, 2, 3D
- supports Rage AMR
- two parallel modes (MPI): mesh replicated, decomposed



IMC quick overview

- particles are
  - emitted by sources (including blackbody material medium)
  - tracked through material medium
    - medium scatters particles: particles deposit energy and momentum
    - material properties discretized on mesh: optical, thermodynamic
    - tallies (energy, momentum dep) accumulated on mesh
- time steps used to linearize radiation-matter interaction
- the particle's track consists of discrete events:
  - collision (scatter, absorption)
  - mesh cell boundary (reflect, cross into new cell, escape)
  - weight cutoff (statistically insignificant, typ. absorbed next step)
  - step end (end of time step, recorded in census for next time step)

Monte Carlo events



collision events

scatter

(Ω_f,ν_f)

(Ω_i,ν_i)

$E_{dep} = h(\nu_i - \nu_f)$

absorption

weight cutoff

(absorbed)

boundary events

reflection

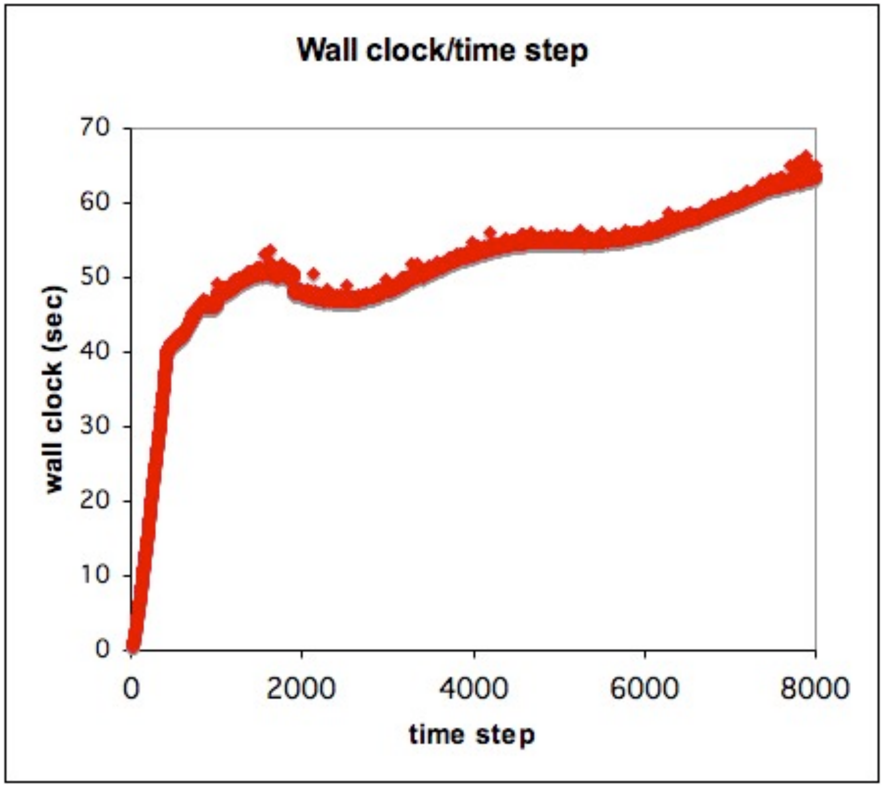
bdy crossing

escape

step end

Milagro performance depends on the problem

- different events take different times to process
  - cutoff event: a few stores
  - effective scatter event: thousands of instructions
- performance also varies within problem
  - number of MC steps & mix of step types changes as materials and fields evolve
- this makes it difficult to summarize performance in a single number



General performance characteristics of the mainline code

- reasonable to excellent cache locality
  - 4-5 9’s L2 hit/instruction
- FP instructions not dominant
  - many integer instructions (e.g. random number generators)
  - branch instructions: ~12%
  - FP: ~18%

Hybrid reimplementation extends production code

- Two approaches:
  - evolve incrementally (“bottom up”)
  - freely rewrite structures & algorithms (“top down”)
- both kept existing large-scale MPI structure on hosts
  - excellent weak scaling
- both decoupled particle generation from particle tracking
- similar performance gains at assessment
- different approaches to work
  - bottom up: Cell SPEs push particles one at a time
  - top down: Opterons generate work packets, any available processor pulls work from queues

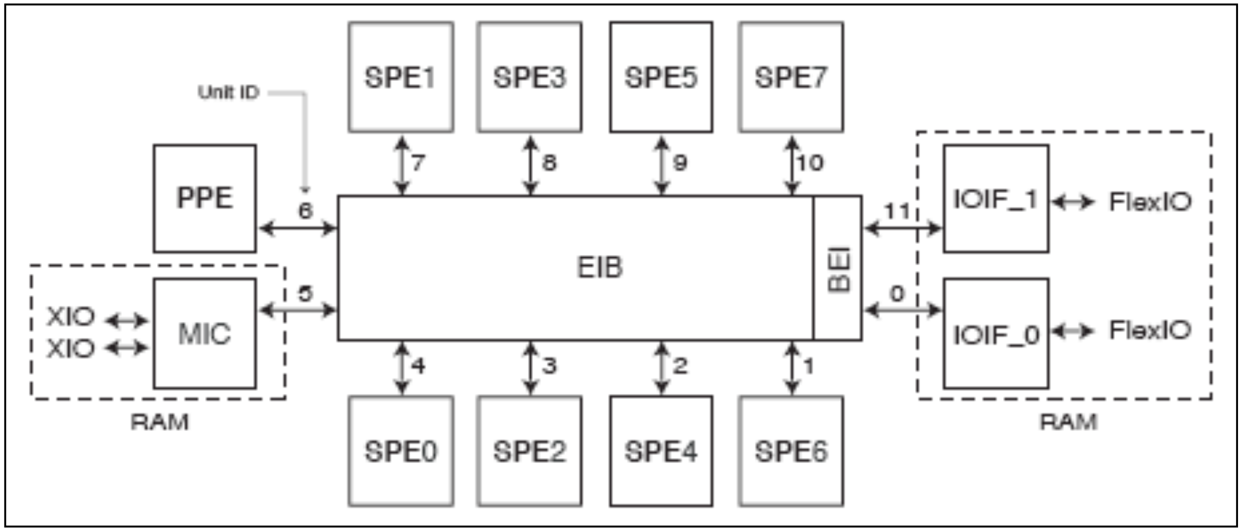
Cell work

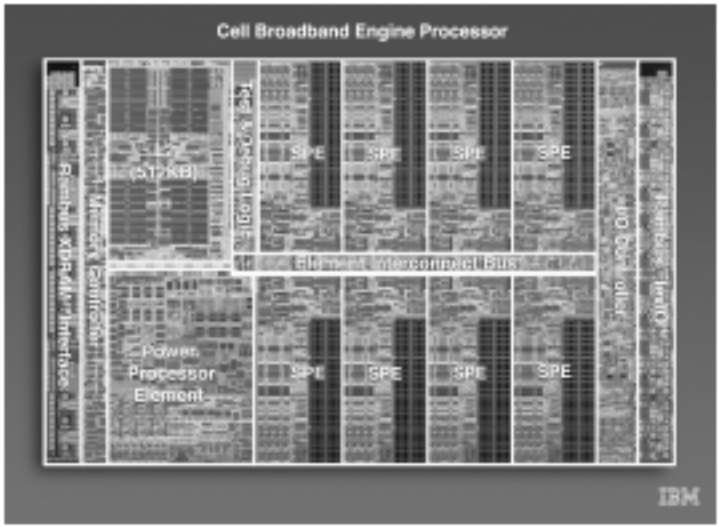
The bottom-up approach started with Cell

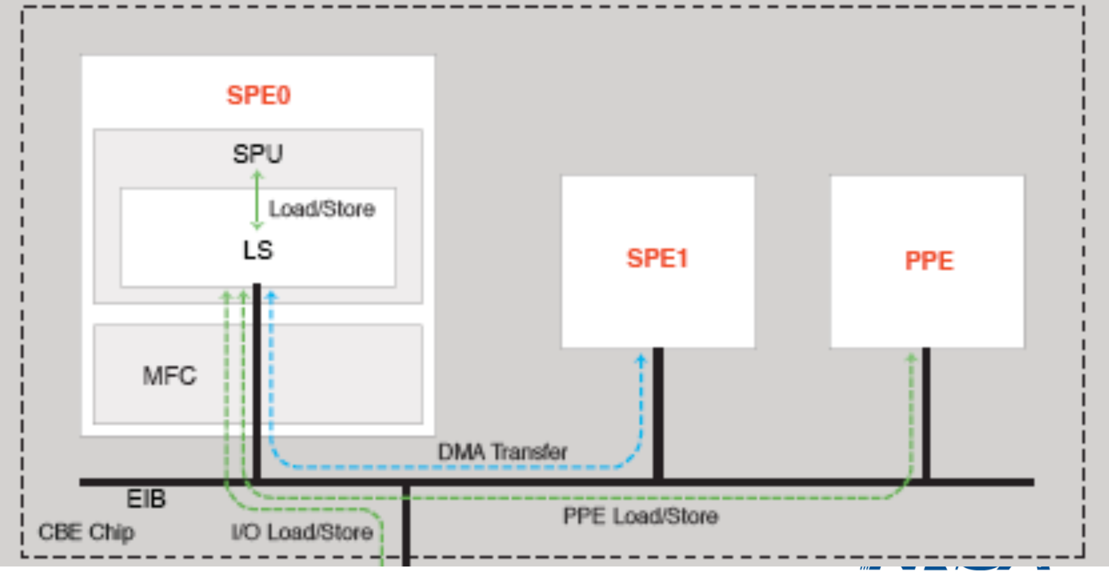
- Cell looked easy compared to other accelerator technologies
- Strategy: use Cell to push particles
  - eventually use Opterons to source particles
  - for tests, a buffer of particles is created, then SPEs push the particles
  - compare to time taken for x86 to push through same buffer
- does this work?
  - can we fit enough code + data into local store?
  - can we make scalar code work on vector machine?
  - correctness: do we get same results as x86?

Cell Broadband Engine: hardware overview

from CBE programmer's Handbook, v. 1.0







Cell Broadband Engine: programmer’s overview

- PPE
  - coordinate SPE work, ~~marshall data for them~~
- SPE
  - vector ISA (no scalar registers, almost all instructions vector)
  - 128 registers x 128 bit
  - limited local store (256 KB code + data)
    - not an automatic cache
  - asynchronous DMA local store ←→ main memory
  - no: cache, OOO, spec. ex., hardware branch predictors
  - no: STL, exceptions, streams
  - tricky: dynamic memory allocation

Cell challenges for Milagro

Worst Cell code ever?

- random access into large data sets
  - opacities (frequency dependent)
  - emission cumulative distribution function (CDF, also freq. dependent)
  - mesh connectivity
  - random, but slowly changing in time
    - mean free path, ~1/σ_a
  - strategy: proxy large data objects on SPE
- scalar code: one particle at a time
  - particles follow different paths, both physical and execution
  - strategy: use vectors, even if half full
- much branching, conditional assignment
  - (that is, if (cond) a = b; else a = c;)
  - strategy: eliminate branches

Cell challenges for Milagro (cont.)

- C++ uncomfortable on SPU
    - exceptions lead to massive bloat
        - Milagro, ClubIMC, Draco use Design By Contract
    - inheritance requires malloc
        - even if you do no dynamic allocation
        - more bloat

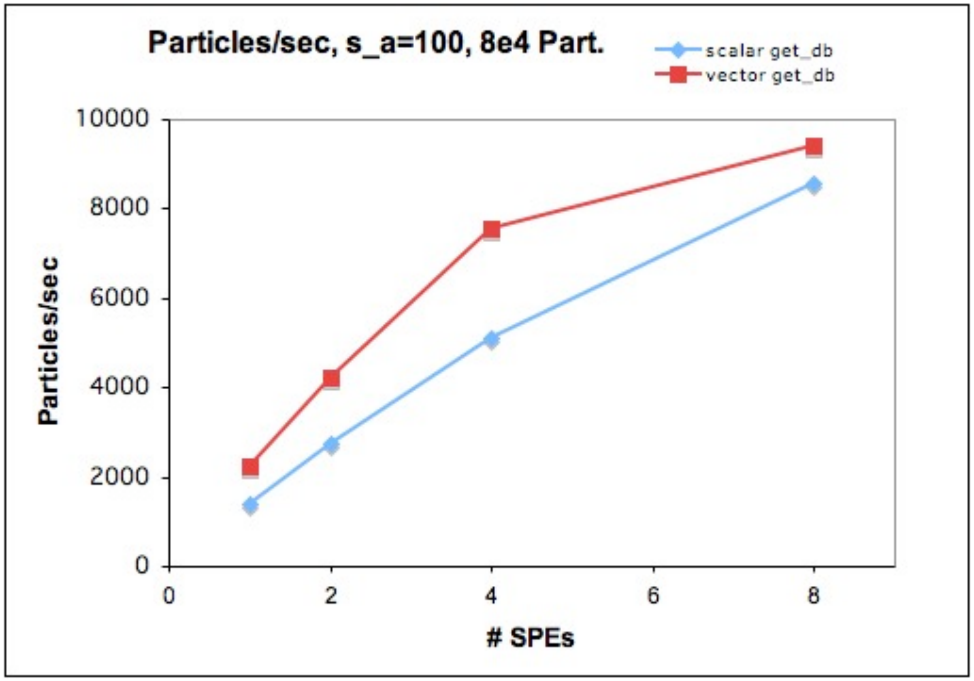
First performance numbers: Nov. 2006

- 8e4 particles, 2 cell 2D RZWedge mesh, dr = dz =1 cm, theta = 5 deg, sig_a = 100 cm^{-1},
- CBE hardware: DD2 @ 2.1 GHz, 512 MB
  - 13.5 s (all 8 SPEs)
  - Cell comm overhead ~ 20%
- x86-64 Xeon @ 3.8 GHz, 2 MB L2, 4 GB
  - ~3.8 s
- allowing for clocks, Cell is 2x slower
  - “Acceleration *is* a vector.” (Jamal, helpful as always)
- ouch!
  - scalars & branching

What went wrong?

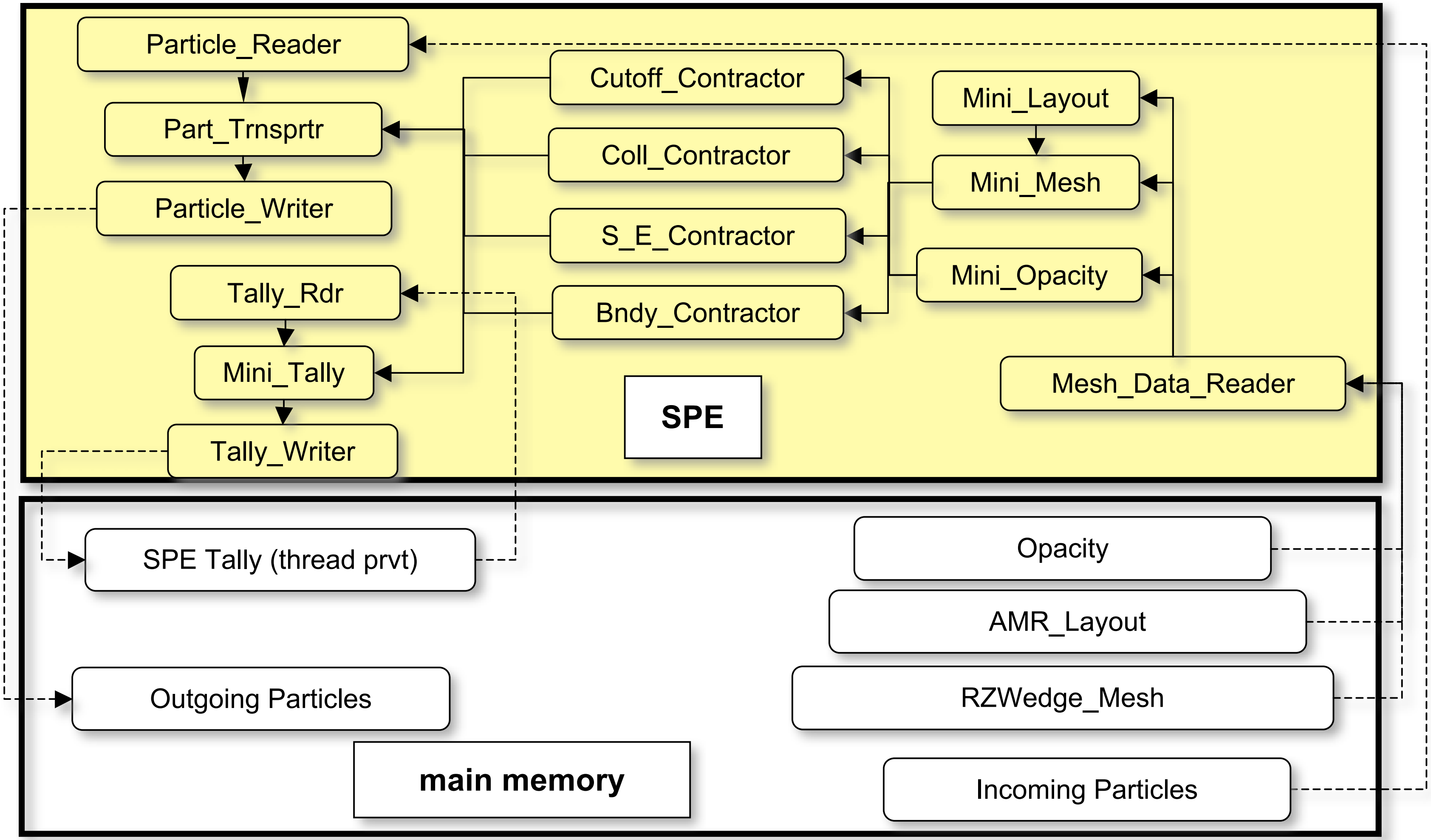
To start with, the PPE was overloaded

- original SPE particle transporter relied on PPE to serialize SPE data
  - mesh connectivity
  - opacity
  - particles
  - vector<vector>
- "this worked fine in the simulator"
- does not scale to 8 SPEs in hardware
- solution: have SPEs fetch data



Proxies represent large data structures in SPE LS

core objects (Contractors) don't know about DMA

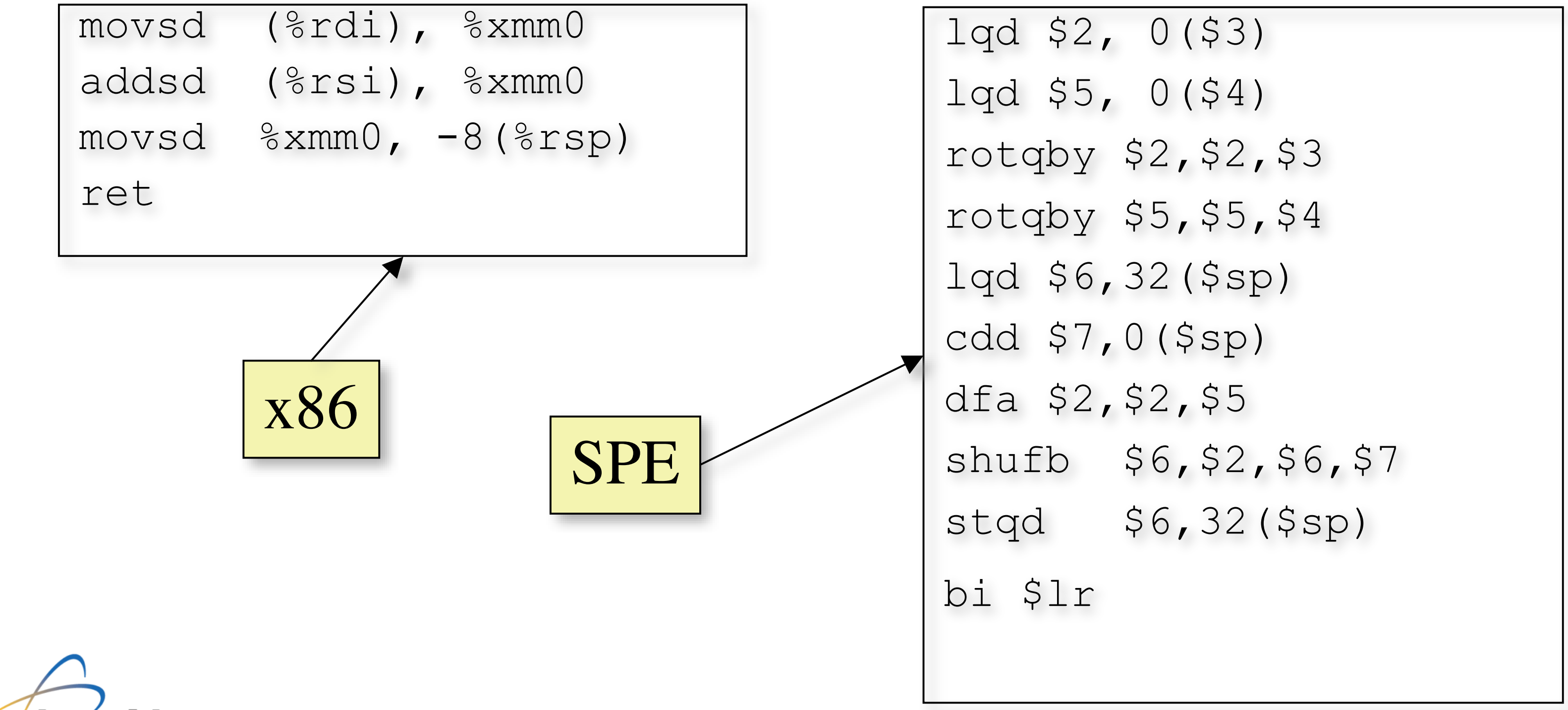


simplified view of data flow

The problem with SPE scalars...

compiler doesn’t know where they are in vector, must shuffle & rotate

```
void add_two( double *a, double *b, double *c){
   *c = *a + *b;        // pointers: compiler doesn’t know alignment of the doubles.
   return;}
```



x86

```
movsd   (%rdi), %xmm0
addsd   (%rsi), %xmm0
movsd  %xmm0, -8(%rsp)
ret
```

SPE

```
lqd $2, 0($3)
lqd $5, 0($4)
rotqby $2,$2,$3
rotqby $5,$5,$4
lqd $6,32($sp)
cdd $7,0($sp)
dfa $2,$2,$5
shufb  $6,$2,$6,$7
stqd   $6,32($sp)
bi $lr
```

The scalar solution: vectorize and “pseudo-vectorize”

- Vectorize when we can
- When we can’t:
  - use half-empty vectors rather than scalars
  - choose other half to avoid NaNs!

Remove branches wherever possible

distance to boundary calculation: 2 examples in 1!

- original (RZ geometry):

```
if Ω[0] != 0.0:
    if Ω[0] > 0.0:   //moving toward face2
       min_dist = (x-face2)/ Ω[0]
        face = 1
    else:
        min_dist = (face1 - x)/ Ω[0]
        face = 2
if Ω[2] != 0.0:
    if Ω[2] > 0.0:
        z_dist = (face6_z)/ Ω[2]
        z_face = 6
    ...
    if( z_dist < min_dist):
        min_dist = z_dist
        face = z_face
if Ω[1] != 0.0:
   ...
```

- Many small, unpredictable branches run inefficiently on Cell SPU
- Original version: 75 decrementer ticks/call (DD2 CBE)
- Called once/MC step--50% of MC step.

Tuning distance-to-boundary calculation

vectorized, branchless version takes 4 ticks--18x faster (*on DD2 CBE)*

- replace conditional assignents with bitmasks, select instructions

```
vec64fp const r0 = sal_splats_64fp( r0);         // {r[0],r[0]}
vec64fp const o0 = sal_splats_64fp( omega0);     // {W[0],W[0]}
vec64fp L12 = sal_sub_64fp( mesh->get_x(), r0); //{face1-r[0],face2-r[0]}
L12 = sal_div_64fp( L12, o0); //{(face1-r[0])/omega[0],(face2-r[0])/omega[0]

// Sets elements to “huge” if o0 is exactly zero
L12 = huge_if_sig_zero_v( L12, o0);

// If omega[0] +ve, passes element 0, sets element 1 to “huge”
L12 = huge_by_sign0_v( L12, o0);
```

... repeat to get L34 and L56 with distances to faces 3,4,5,6, filtered by direction cosine
... actual code is interleaved

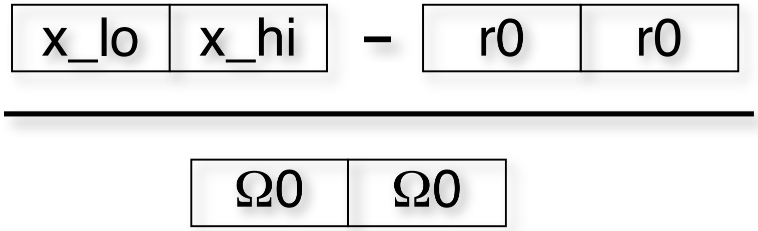
```
// select least of distances to face 1,2,3,4,5,6
vec_double2 m1       = sal_min_64fp( L34, L12);
vec_double2 min_dist = sal_min_64fp( L56, m1);
min_dist = sal_min_64fp( min_dist, sal_flip_64fp( min_dist)); // { min,m
...
```

Tuning distance-to-boundary calculation

1: compute distance to low-x and high-x faces (1&2)

- replace conditional assignents with bitmasks, select instructions

```
vec64fp const r0 = sal_splats_64fp( r[0]);
vec64fp const o0 = sal_splats_64fp( omega0);
vec64fp L12 = sal_sub_64fp( mesh->get_x(), r0);}
L12 = sal_div_64fp( L12, o0);
```



x_lo x_hi – r0 r0

Ω0 Ω0

Tuning distance-to-boundary calculation

2: if moving parallel to x, set both values to “huge”

```
L12 = huge_if_sig_zero_v( L12, o0);
```

```
inline vec64fp huge_if_sig_zero_v( vec64fp const source,
                                   vec64fp const signifier){
vec64fp const signif_abs = sal_and_64fp( signifier, compl_sign_bits);
vec64ui const comp  = sal_cmpeq_64ui( zero_v, signif_abs);
return spu_sel( source, huge_v, comp);
}
```

Ω0 == 0.0 ?	10^{30}	10^{30}	:	dist to lo-x	dist to hi-x

Tuning distance-to-boundary calculation

3: if headed toward low-x, set dist to high-x to “huge”

```
L12 = huge_by_sign0_v( L12, o0);
```

```
inline vec64fp huge_by_sign0_v( vec64fp const source,
                                vec64fp const signifier){
// Extract sign bit of low element of signifier
vec64ui const sign = sal_and_64fp( signifier, sign_bit0);
// Spread the sign bits across the remaining 127 bits
vec32ui sign4 = spu_rlmaska( sign, -31);
sign4 = spu_shuffle( sign4, sign4, low_word_shuff_pattern);
return spu_sel( source, huge_v,
                sal_xor_64fp( huge_by_sign_pattern, sign4));
}
```

$\Omega 0 > 0.0$? | 10^{30} | dist to hi-x | : | dist to lo-x | 10^{30} |

The Cell performance journey begins...

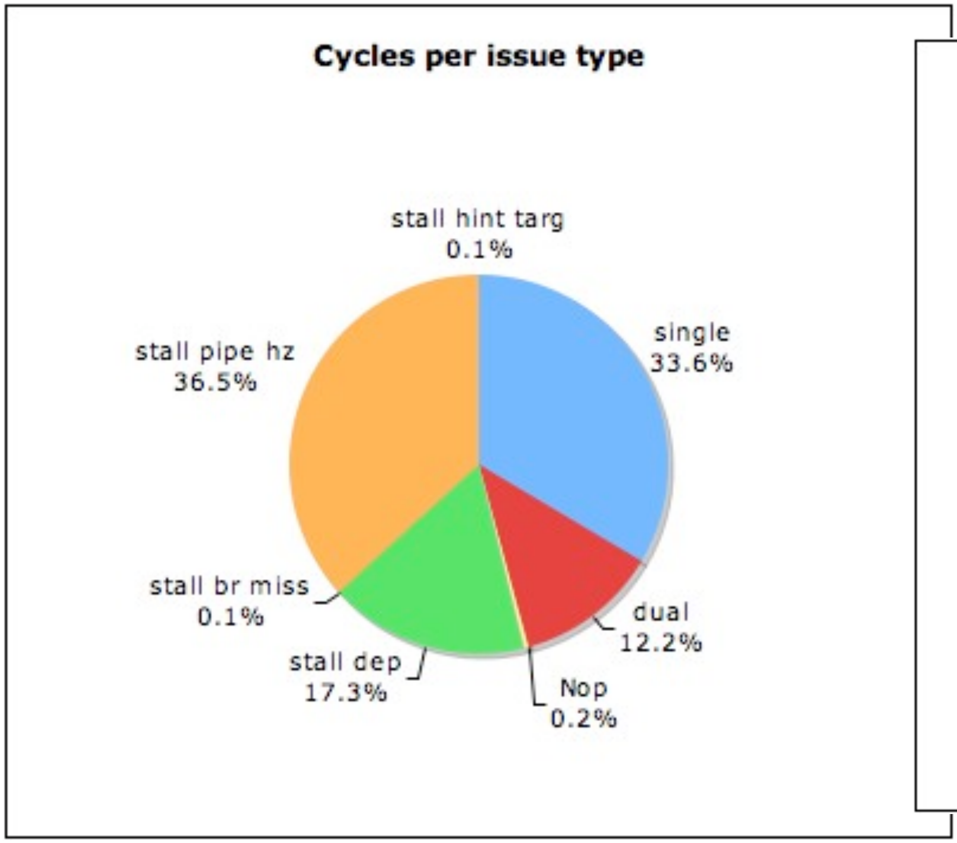
Note: test problem & measurement changed around 2/1/07

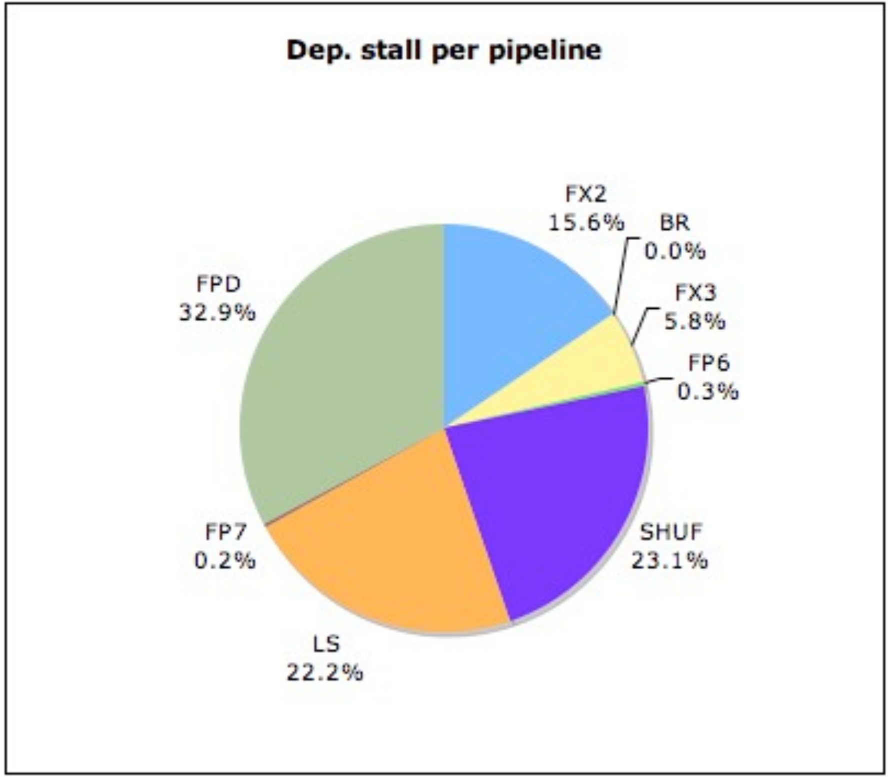
Date	Improvements	Speedup	rel. to
11/2/06	it's alive! (barely)	0.5*	Xeon
1/4/07	SPEs handle tallies, fetch per-cell data	0.89*	Xeon
2/5/07	read/write multiple particles, vect. RNG;	1.9	Xeon
2/13/07	vect. bidding, opacity; improve group search	3.5	Flashd
2/14/07	eliminate one branch in sample group	5.0	Flashd
2/20/07	debranch apply_ew; vect. ew & fraction, Tally, sample_dir	6.0	Flashd
2/22/07	vect. RNG; inline vector RNG	6.9 6.2	Flashd RR base
3/5/7	thread private tallies (eliminate last PPE-mediated task)	2.8, optically thin problem	RR base

*adjusted by clock speeds (measured on 2.1 GHz CBE).

Cell performance characteristics

Effective scatter MC step (measured in simulator 2/22/07)





Cell SPE transport modifications: summary

- rewrote
  - contractors
  - streaming & scattering operators
  - particle transporter
  - multigroup particle classes
- added
  - communications classes
  - thread private tallies

Other mistakes (some of them, anyway)

- Worked for first 4-5 months on simulator
  - early hardware couldn’t accomodate EC code; cross-compile not working
  - spent lots of time perfecting cache scheme that overloaded the PPE
  - lots of race conditions hidden until I worked on hardware
  - nonetheless, simulator is a very valuable tool, esp benchmarking.
- SPE memory and compute really are asynchronous
  - DMAs from stack buffers can be ruined when that memory is reused in another function call (DMA from global buffers)
  - DMA can start before a store completes
    - use spu_dsync to force all loads/stores to complete before DMA starts
- And the mother of all race conditions...

Ouch

Can you spot the race condition?

After a modification that made the whole SPE code run a little faster, I hit a strange race condition. About once every 10^5 passes through a section of code, the program would hang.

```
volatile vec_uint4 flags[4] __attribute__((aligned(16)));
DMA to/from buffer 0 with tag 0
Fenced DMA 4 bytes into flags[0] with tag 0 (ensures completion)
DMA to/from buffer 1 with tag 1
Fenced DMA 4 B into flags[1] with tag 1
while(flags[0] != 0xaaaaaaaa){}  // now ok to use buffer 0
flags[0] = 0;  // clear guard for next use
... do work on buffer 0 ...
while(flags[1] != 0xaaaaaaa){}  // occasionally hangs here. Why?
flags[1] = 0;
```

Every operation works on a vector, even when the code says it worked on a scalar.

flags[0] = 0; // this does not operate on flags[0] only!

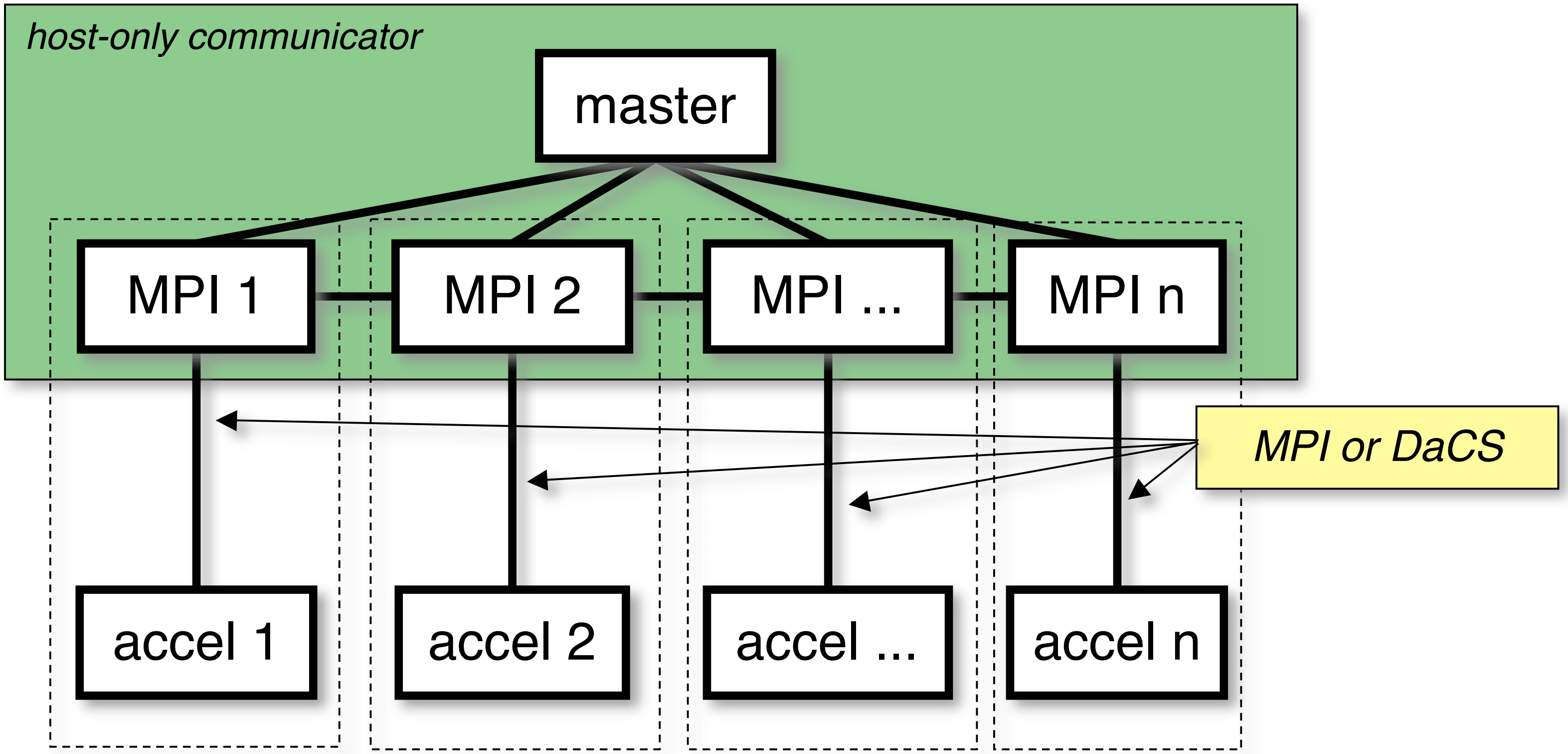
- Load 16 bytes from address of flags[0];
- shuffle 4 bytes of 0 into the first four bytes of the vector;
- Store 16 bytes back to address of flags[0]
  - flags[1], flags[2], and flags[3] are the same as they were when loaded.
- If the second fenced DMA--into flags[1]--arrived up in flags[1] between the time we loaded and the time we stored flags, the store operation will overwrite the new value (0xaaaaaaaa) with the old value (0). It's as if the DMA never happened.
- Obvious solution: put the flags in separate quadwords.

Move to hybrid

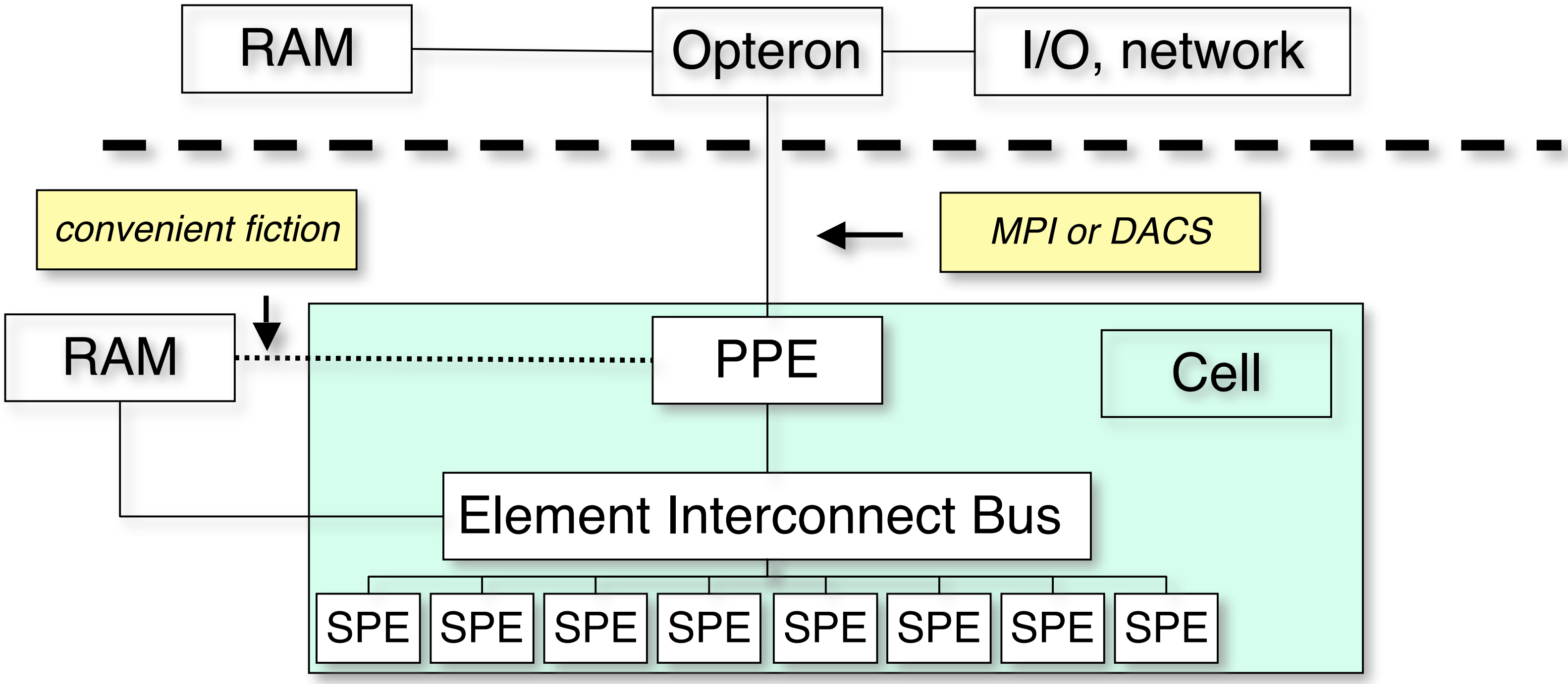
Move to hybrid

- demonstrated Cell performance
  - with additional known reserves of performance improvements
- begin to program for Hybrid Roadrunner
- integrate Cell particle push into hybrid MPI application

Hybrid IMC communication structure



An app programmer’s view of Roadrunner hybrid node



A bottom-up hybrid time step:

who does what in each phase of a time step

	Opteron: host	PPE: manager	SPE: worker
initialize	▪ signal Cell to begin time step ▪ compute mesh, opacity ▪ send mesh, opacity to Cell	▪ receive mesh/opacity ▪ start SPE threads	▪ wait
transport	▪ generate particles, send to PPE ▪ recover spent particles from PPE, retire them (kill, census)	▪ synchronize particle I/O between host & workers	▪ load particles, mesh, opacity, tally data ▪ transport particles -- refresh mesh, tally opacity data ▪ store particles, tallies
finalize	▪ signal Cell ▪ wait for Tally finished signal ▪ recover Tally, update material state	▪ join SPE threads ▪ merge thread-private tallies ▪ signal host	▪ idle

Bottom-up IMC design points

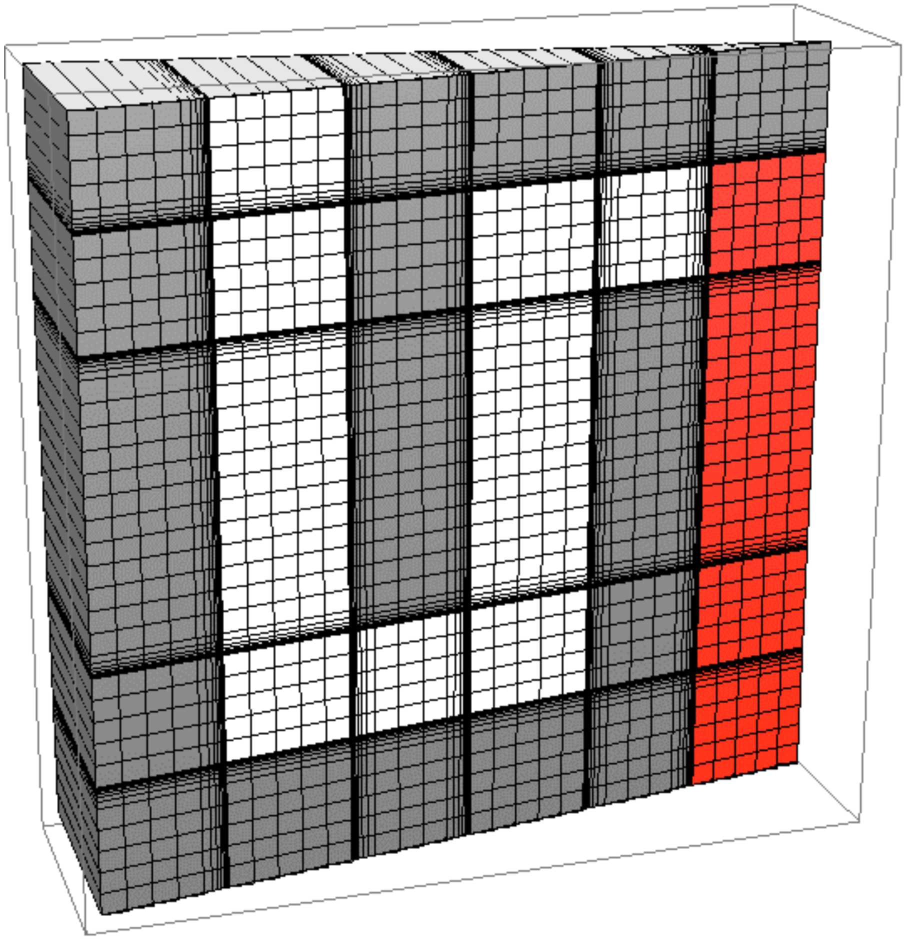
- abstract accelerator interface for host
  - host provides mesh & opacity, input & output particle streams
  - accelerator proxies send particles to/from accelerator
    - encapsulates host-side particle communication
    - option: do light workloads in a different thread on the Opteron (not implemented)
- on accelerator (Cell PPE), simple event loop moves particles in/out, synchronizes with workers (SPEs)
- SPEs
  - load particles, mesh & optical data, tally data
  - transport particles
  - store spent particles and tally data.

Bottom-up IMC implementation

- 2D AMR RZWedge mesh, multigroup frequency, mesh replicated
  - most challenging in production at time of outset
  - challenging to program, challenging for Cell
  - parallel scaling properties unchanged from mainline code
  - particles move in 3D
  - did not implement random walk, surface tallies, mat. advection
  - did not implement domain-decomposed
- revised particle transport code for Cell SPE
  - bring subsets of large data sets to SPE local store
  - partially vectorized
- for host, rewrote IMC_Manager class, Transporter classes, new classes for accelerator communications.

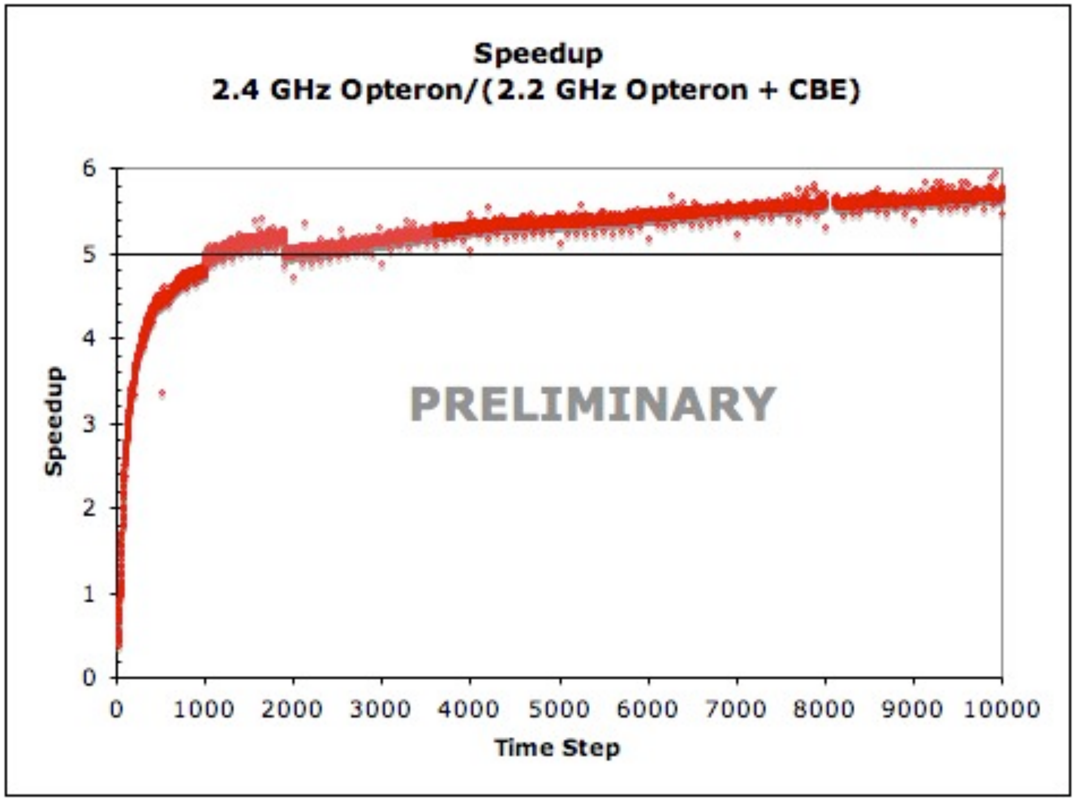
Test problem “double bend”

- problem specs
  - RZWedge mesh (10 deg)
  - volume source
  - material initially cold
  - ratio-zoned boundary layers
  - regular cells: 0.1 x 0.1 cm
    - 20 m.f.p.^{2}/0.02 m.f.p.^{2}
  - 100 freq. groups
  - 5e5 particles/time step
    - ~1 particle/phase space element
  - 10e4 time steps, 0.01 sh



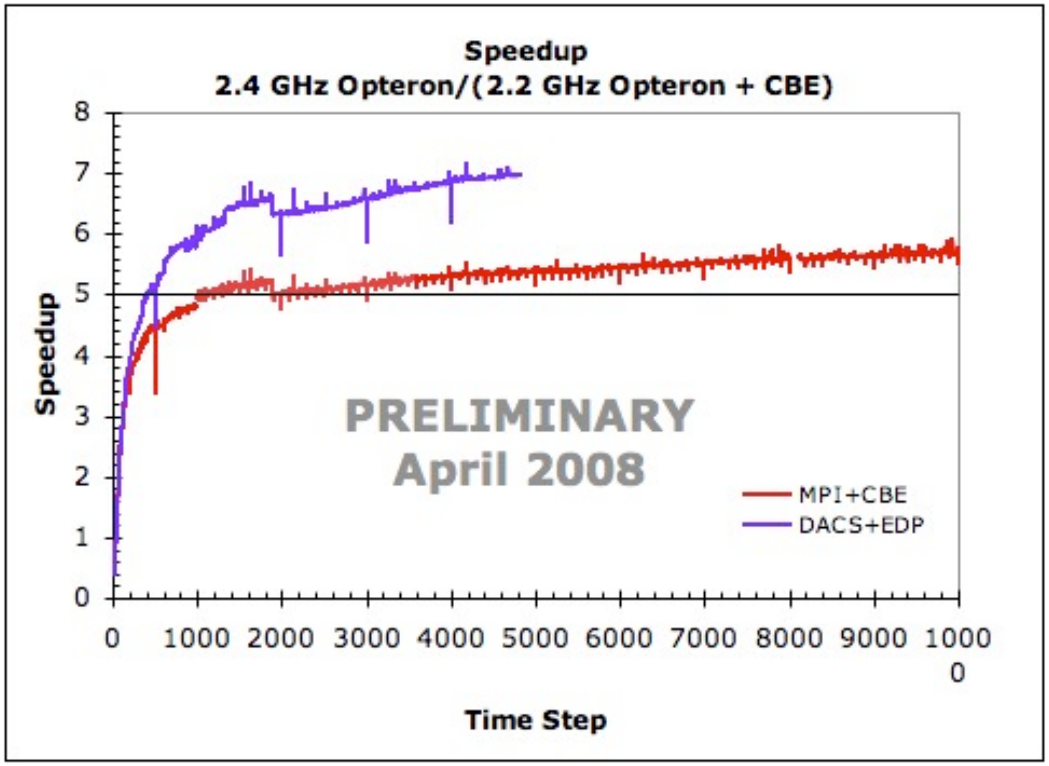
double bend problem > 5x faster at assessment

- Opteron + CBE vs Opteron alone
- Measured with current gen. Cell
- Roadrunner cells will have improved double precision
- time-to-solution for full hybrid code out to 100 sh.



Assessment code on EDP/PCIe

- Motive: fallback for SN light curve open science runs
- Changed code to run on new hardware
  - MPI-->DACS
  - libspe-->libspe2
- Enhanced double precision
  - “PowerXCell8i”
  - no issue stall
  - 9 cycle DP latency
- Very preliminary
  - as in, yesterday
  - still checking solution quality



“Proven reserves” of performance improvements

- Assessment code did not use vector RNG
  - creates problem with reproducibility
  - can be solved
- Flatten data structures
  - simplify refresh on mesh cell bdy crossing
    - 3 serial DMAs -> 1 DMA
    - apply_event 2x faster
  - important for optically thin case

Further work

- reconcile approaches
- full vector MC?
- transform to production code
  - research code--lots of ugliness
  - tests, tests, tests
  - need to incorporate surface tracking, material motion
  - maintain platform agnostic version
    - reference implementation for methods R&D, next transition
  - create short-vector ISA-agnostic version (SAL)
    - same code, different compilers for SSE, Altivec, SPE
- support from RR to update research code
  - look at performance with EDP v CBE, DACS v MPI, libspe2 v libspe

What is and is not hard about Roadrunner?

- not so hard:
  - compilers, thread mgmt, byte swapping, etc.
    - do it once, forget it once
  - SPU intrinsics: if you can use functions, you can use intrinsics
    - better yet, use SAL (in the not-too-distant-future)
- hard part is data:
  - good compute depends on data layout
    - how different is that from writing cache friendly code?
  - full vectorized MC tough
    - independent execution paths
- vector ISA is medium--takes, rewards bit of work
  - how different is that from SSE, Altivec?
    - cleaner than SSE, anyway
- hybrid, heterogeneous programming is newish--MPMD

SIMD Abstraction Layer (SAL)

- Provide uniform interface to SIMD types, intrinsics, & lib calls
  - SPE, SSE, Altivec, scalar
  - uniform register model
- Where ISA’s overlap, provide direct access to instructions/intrinsics
  - e.g. add, mul, etc
- Where they don’t, provide task-oriented calls
  - efficient implementation under the covers
  - e.g. instead of SPE’s shuffle, provide specific permutation tasks
  - but we’ll provide a shuffle, too
- Status: rickety alpha
  - about 40 intrinsics wrappers, 2-6 types @; about 5 lib calls
  - many still implemented using scalar instructions
  - testing is sporadic
  - inline C functions in header files; overloaded C++ wrappers in progress
  - friendly users welcome to take a look (LANL SF)